Learning Continuous Control Agents with Evolutionary Algorithms

Author: `Andrew Malta`

Introduction

In this project, I explored the application of evolutionary algorithms to learn policies for continuous control agents in the OpenAI Gym. The OpenAI Gym, while traditionally used to measure the performance of reinforcement learning approaches on toy environments from Atari games to motion tasks, provided me with an ideal framework to test the performance of evolutionary algorithms on learning deterministic policies for agents. I first got interested in this topic when I read OpenAI's paper, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning", which experimentally demonstrated that it is possible to rival the performance of reinforcement learning in many tasks while gaining the desirable property of highly parallelizable training procedures. This report along with an excellent blog post about applying these techniques to the OpenAI gym in particular, both referenced at the bottom, encouraged me to try my hand at some of the hardest control tasks offered in the OpenAI gym.

Environments

In choosing an environment to experiment with, I first looked for a task that would be particularly amenable to reinforcement learning, as I wanted to see that a typical reinforcment learning task could be solved through the use of an evolutionary algorithm. I settled on the Bipedal walking task in the OpenAI gym as it seemed to meet these criteria and due to the fact that it had both a normal and a hadcore version, which I thought I might be able to exploit in the training process. Very simply, the goal of the task is to have your agent navigate itself across the terrain in front of it and reach the goal on the other end. The reward function penalizes stumbling, defined to be when the head hits the ground, and the use of excess motor torque. A good solution is robust to the terrain, but does not use too much motor torque to maintain this robustness.

The input to the model is a stream of 24 dimensional vectors each of which includes hull angle speed, angular velocity, horizontal speed, vertical speed, position of joints and joints angular speed, leg contacts with the ground, and 10 lidar inputs that encode objects in the vicinity. The output is a 4 dimensional vector representing the amount of torque to apply to each of the 4 joints on the agent.

The normal version of the environment has small random variation in the terrain, but all in all it is pretty flat.

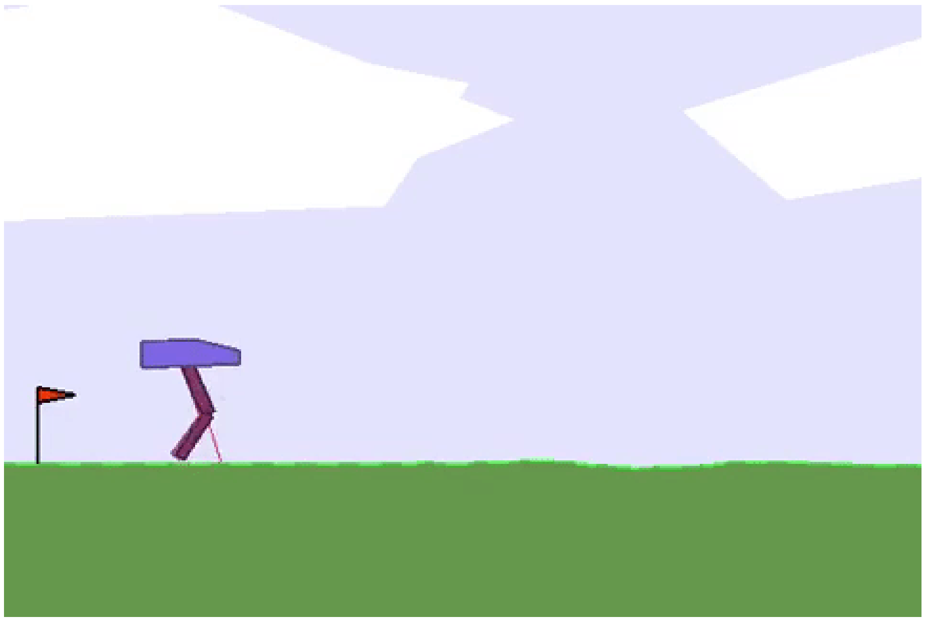

Figure 1: Alt Text

The hardcore of the environment, on the other hand, has pits, ramps, and obstructions of various sizes that the agent has to navigate. Due to this terrain, it is much easier for the agent to get stuck in local minima, as it can learn how to navigate some of the terrain but not the rest.

The hardcore task poses a major leap in difficulty as the terrain varies much more significantly for each run of the environment than the normal version of the environment does. This forces you to learn a robust policy to work in any configuration of these ramps, pits, and obstructions rather than learn a particular motor sequence that works for a particular environment. This was not a major issue in the normal bipedal walking environment as the terrain variation was minimal, allowing the evolutionary algorithm to simply learn a motor sequence that propelled itself forward while keeping its balance.

Model

For this project, I decided to roll my own small neural network framework as I wanted to have direct access to the parameterization of the weights and biases of the network through a single flat vector. This, in my mind, was the easiest way to interface the evoltionary algorithms with my model as most libraries make you go out of your way to do something other than the classic back propagation with some optimizer. I wanted to keep the model simple because I wanted to

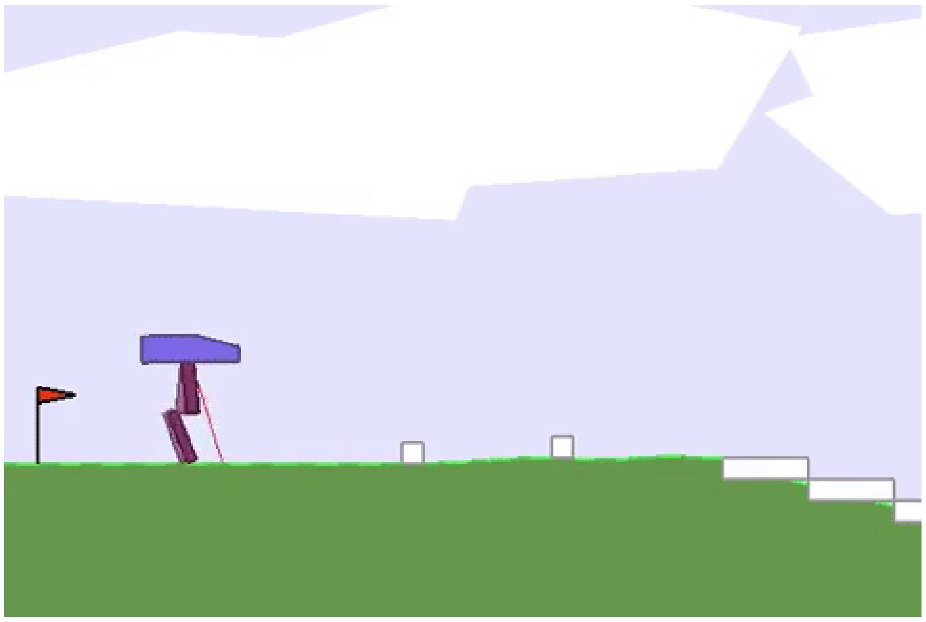

Figure 2: Alt Text

try keep the focus on the optimzation procedure, and I wanted the optimzation routine to run quickly.

The model I ended up choosing was a feed-forward nerual network with 2 hidden layers and hypberbolic tangent activation, which ended up totaling a parameter vector of length 2804 when you account for the biases. This number of parameters, of course, was a main concern when I was choosing the size of my model, as I wanted to ensure that the neural network was able to represent the function I was trying to approximate. That being said, I wanted to keep the parameter vector small enough for performance reasons, as I wanted to test the performance of the CMA-ES optimization algorithm, which scales poorly with the size of the parameter vector. The code for this simple neural network model is listed below:

```
import numpy as np

class Net():
  def __init__(self, weights):
    # an affine operation: y = Wx + b
    self.dimensions = [(24, 40), (40, 40), (40, 4)]
    self.layers = []
    self.biases = []
    if not weights is None:
      tmp = 0
      for d in self.dimensions:
        length = np.prod(d)
        self.layers.append(np.reshape(weights[tmp: tmp + length], d))
        tmp += length
        self.biases.append(np.reshape(weights[tmp: tmp + d[-1]], (1, d[-1])))
        tmp += d[-1]

  def set_model_params(self, weights):
    self.layers = []
    self.biases = []

    tmp = 0
    for d in self.dimensions:
      length = np.prod(d)
      self.layers.append(np.reshape(weights[tmp: tmp + length], d))
      tmp += length
      self.biases.append(np.reshape(weights[tmp: tmp + d[-1]], (1, d[-1])))
      tmp += d[-1]

  def forward(self, x):
    working_tensor = x
    for i in xrange(len(self.layers)):
      affine = np.dot(working_tensor, self.layers[i]) + self.biases[i]
      working_tensor = np.tanh(affine)
    return working_tensor[0]

  def num_flat_features(self):
    ret = 0
    for d in self.dimensions:
      ret += np.prod(d)
      ret += d[-1]
    return ret
```

Optimization

I played around with a number of evolutionary algorithms, some of which do not have a canonical name and others that certainly do. When I was performing experiments on some other environments, that are not described in detail in this report, I was still not using the MPI interface I adapted from ESTool.

Ask/Tell Interface

One design pattern that I found incredibly useful both to think about the evolutionary algorithms conceptually, and to implement them in python was the ask/tell interface that was described in the blog post about ESTool. Conceptually, it is very simple. The algorithm exposes an **ask** function that will supply the user with new parameter vectors that are sampled in some way from the current state of the algorithm. The user then takes these new parameter vectors and computes the fitness score for each of them. After the user completes the fitness evaluations, the user calls the algorithms **tell** function, which supplies the algorithm with the fitnesses of the parameter vectors that it supplied the user. The algorithm then uses this information to update any internal state of the algorithm to inform the next iteration of the ask/tell interface.

Outside of improvements to code readability, this paradigm allows the programmer to abstract away how the fitnesses are being computed. More specifically, it allows the programmer to decide if they want to parallelize this computation, which they often might, without having the parallelization code to have to touch the implementation of the optimization routine.

WinnerES

WinnerES is a basic template for a evolutionary algorithm with a simplified update step of just choosing the best child from the generation and using it as the starting point of the next generation. This is not great for a number of reasons, but it could work for some simple tasks as I saw. In other words this can be thought of as a randomized sampling of directions on the objective function to approximate the gradient at your current parameter vector. The larger your generation size, n, the better your approximation of the gradient gets, but the longer the computation takes per step.

Pseudocode for WinnerES

```
%%latex

\begin{align}
    &\theta_{0} = \vec{0} \\
    &\text{for some number of generations:} \\
        &\hspace{1cm} \epsilon_{1,...,n} \sim N(\sigma, I) \\
        &\hspace{1cm}  j = \text{argmax}_{i} f(\theta_t + \epsilon_i) \\
        &\hspace{1cm}  \theta_{t+1} = \theta_{t} + \epsilon_j \\
\end{align}
```

$$\theta_0 = \vec{0} \tag{1}$$
for some number of generations: (2)
$$\epsilon_{1,\ldots,n} \sim N(\sigma, I) \tag{3}$$
$$j = \text{argmax}_i f(\theta_t + \epsilon_i) \tag{4}$$
$$\theta_{t+1} = \theta_t + \epsilon_j \tag{5}$$
(6)

Implementation of WinnerES

```
class Winner_ES:
    def __init__(self, kwargs):
        self.theta_dim = kwargs["theta_dim"]
        self.gen_size = kwargs["gen_size"]
        self.sigma = kwargs["sigma"]
        self.theta = np.zeros(self.theta_dim)

    def ask(self):
        self.epsilon = np.random.randn(self.gen_size, self.theta_dim)
        self.solutions = self.theta.reshape(1, self.theta_dim) + self.epsilon * self.sigma

        return self.solutions

    def tell(self, fitnesses):
        argsort = list(np.argsort([-val for val in fitnesses]))
        self.theta = self.solutions[argsort[0]]

    def result(self):
        return self.theta

    def set_theta(self, theta):
        self.theta = np.array(theta)
```

Single Threaded OpenES

The next algorithm I implemented and tested on some toy examples was the non-parallelized version of OpenAI's evolutionary strategy, which is described in algorithm 1 in the paper in the references. In my code I reference it as WeightedAverageES, and the updates are as follows:

Pseudocode for Single Threaded OpenES

```
%%latex

\begin{align}
    &\theta_{0} = \vec{0} \\
    &\text{for some number of generations:} \\
      &\hspace{1cm} \epsilon){1,...,n} \sim N(\sigma, I)\\
      &\hspace{1cm} f_i = f(\theta_t + \epsilon_i) \\
      &\hspace{1cm} \theta_{t+1} = \theta_t + \frac{\alpha}{n\sigma} \sum_{i}^n f_i * \epsi
\end{align}
```

$$\theta_0 = \vec{0} \tag{7}$$

$$\text{for some number of generations:} \tag{8}$$

$$\epsilon)1, ..., n \sim N(\sigma, I) \tag{9}$$

$$f_i = f(\theta_t + \epsilon_i) \tag{10}$$

$$\theta_{t+1} = \theta_t + \frac{\alpha}{n\sigma} \sum_{i}^{n} f_i * \epsilon_i \tag{11}$$

$$\tag{12}$$

Implementation of Single Threaded OpenES

```python
class Weighted_Average_ES:
    def __init__(self, kwargs):
        self.theta_dim = kwargs["theta_dim"]
        self.gen_size = kwargs["gen_size"]
        self.sigma = kwargs["sigma"]
        self.alpha = kwargs["alpha"]
        self.n = kwargs["n"]
        self.theta = np.zeros(self.theta_dim)

    def ask(self):
        self.epsilon = np.random.randn(self.gen_size, self.theta_dim)
        self.solutions = self.theta.reshape(1, self.theta_dim) + self.epsilon * self.sigma

        return self.solutions

    def tell(self, fitnesses):
        argsort = list(np.argsort([-val for val in fitnesses]))
        fitness_update_sum = np.zeros(self.theta_dim)

        for i in argsort[0:n]:
            fitness_update_sum += (self.epsilon[i] * fitnesses[i])

        self.theta = self.theta + (self.alpha / self.n * self.sigma) * fitness_update_sum

    def result(self):
        return self.theta

    def set_theta(self, theta):
        self.theta = np.array(theta)
```

Note that this algorithm is very similar in spirit to the basic evolutionary algorithm, WinnerES; however, rather than just updating the internal parameter vector θ to be the best performing member of the population, it performs a weighted average of the best n members of the populated weighted by their corresponding fitness score.

CMA-ES (Covariance Matrix Adaptation)

CMA-ES is one of the more famous gradient-less optimization routines for non-convex optimization problems. It too, is an evolutionary algorithm; however it has significantly more bells and whistles than previously presented algorithms. The most salient difference, from which CMA gets its name, is the way that the algorithm handles pairwise interactions between parameters in vector it optimizing over. It operates by attempting to maximize the liklihood of sampling previously good candidates and fruitful search steps. This algorithm does not scale extremely well with the length of the parameter vector, as one of the update steps involves inverting the covariance matrix C. As of now, the best known algorihtms for computing an inverse of a matrix runs in $O(n^2.373)$ time, which will give us trouble when we approach n of about 10,000.

Pseudocode for CMA-ES

Image Source: https://en.wikipedia.org/wiki/CMA-ES#Algorithm

For a more in-depth description of the individual update steps, my implementation follows the implementation details of the algorithm described on Wikipedia at the link above.

Implementation of CMA-ES

```
class CMA_ES:
    def __init__(self, kwargs):
        self.theta_dim = kwargs["theta_dim"]
        self.gen_size = kwargs["gen_size"]
        self.start_sigma = kwargs["start_sigma"]
        self.mu = kwargs["mu"]

        self.theta = np.zeros(self.theta_dim)

        self.n = self.theta_dim  # parameter for ease of readability
        self.lamb = self.gen_size # use same notation as wikipedia

        # initialization of the 5 major parameters that we update
        self.mean = np.zeros(self.n)
        self.sigma = self.start_sigma
        self.C = np.identity(self.n)
        self.p_sigma = np.zeros(self.n)
        self.p_c = np.zeros(self.n)

    #TODO: Vectorize this code for perf
    def ask(self):
        self.solutions = []
        for _ in xrange(self.gen_size):
            self.solutions.append(np.random.multivariate_normal(
                self.mean, self.sigma * self.sigma * self.C))

        return self.solutions

    def tell(self, fitnesses):
        tmp = self.mean
        sorted_indices = np.argsort(fitnesses)

        # update m
        self.mean = 0
        # using equal weighting
        for i in sorted_indices[0:self.mu]:
            self.mean += self.solutions[i] / self.mu

        # update p_sigma
        c_sigma = 3. / self.n
        df = 1 - c_sigma # discount factor
        C_tmp = np.sqrt(np.linalg.inv(self.C))
        displacement = (self.mean - tmp) / float(self.sigma)

        self.p_sigma = (df * self.p_sigma +
            np.sqrt(1 - np.power(df,2)) * np.sqrt(self.mu) * C_tmp * displacement)

        # update p_c
        c_c = 4. / self.n
        df2 = (1 - c_c)
        alpha = 1.5

        norm = np.linalg.norm(self.p_sigma)
        indicator = norm >= 0 and norm <= alpha * np.sqrt(self.n)
        complements = np.sqrt(1 - np.power((1 - c_c), 2))
        neutral_selection = np.sqrt(self.mu) * displacement
        self.p_c = df2 * self.p_c + indicator * (complements * neutral_selection)

        # update Covariance Matrix
        c_1 = 2. / np.power(self.n, 2)
        c_mu = self.mu / np.power(self.n, 2)
        c_s = (1 - indicator)* c_1 * c_c * (2 - c_c)

        tmp_pc = self.p_c.reshape(-1, 1)
        rank_1_mat = c_1 * tmp_pc.dot(tmp_pc.transpose())

        mat_sum = np.zeros(self.C.shape)
        for i in xrange(self.mu):
            vec = ((self.solutions[sorted_indices[i]] - tmp) / self.sigma).reshape(-1, 1)
            mat_sum += (1. / self.mu) * vec.dot(vec.transpose())

        self.C = (1 - c_1 - c_mu + c_s) * self.C + rank_1_mat + c_mu * mat_sum

        # update sigma
        d_sigma = 1  # dampening parameter
        p_sigma_norm = np.linalg.norm(self.p_sigma)
        tmp = (1 - 1. / (4 * self.n) + 1. / (21 * self.n * self.n))
        nse = np.sqrt(self.n) * tmp  # neutral selection expectation
        self.sigma = self.sigma * np.exp((c_sigma / d_sigma) * ((p_sigma_norm / nse) - 1))

    def set_theta(self, theta):
        self.mean = np.array(theta)

    def result(self):
        return self.mean
```

OpenES

Lastly, I implemented OpenAI's OpenES algorithm, with some small additions that are common to evolutionary algorithms, such as a weight decay, an adaptive learning rate, and adaptive standard deviation for the mutation rate. The basic algorithm is the same as we described before, however there is some added complexity in setting up the algorithm to work efficiently under the parallel setting. This is the algorithm that I was able to get quick convergence with to reasonable solutions to the somewhat high dimensional Bipedal Walking environment.

Implementation of an Extended OpenES

```
''' helper functions taken from OpenAI's implementation of their algorithm which
can be found here:
(https://github.com/openai/evolution-strategies-starter/blob/master/es_distributed/es.py)
'''
def compute_ranks(x):
    """
    Returns ranks in [0, len(x))
    Note: This is different from scipy.stats.rankdata, which returns ranks in [1, len(x)].
    """
    assert x.ndim == 1
    ranks = np.empty(len(x), dtype=int)
    ranks[x.argsort()] = np.arange(len(x))
    return ranks

def compute_centered_ranks(x):
    """
    https://github.com/openai/evolution-strategies-starter/blob/master/es_distributed/es.py
    """
    y = compute_ranks(x.ravel()).reshape(x.shape).astype(np.float32)
    y /= (x.size - 1)
    y -= .5
    return y

def compute_weight_decay(weight_decay_coef, theta):
    """
    compute the weight decay for the model parameters. Used to encourage the
    mean model parameter squared to not get too large.
    """
    flattened = np.array(theta)
    mean_sq_theta = np.mean(flattened * flattened, axis=1)
    return -weight_decay_coef * mean_sq_theta

class OpenES:
    ''' Basic Version of OpenAI Evolution Strategy.'''
    def __init__(self, args):
        self.theta_dim = args["theta_dim"]
        self.sigma = args["sigma_start"]
        self.sigma_init = args["sigma_start"]
        self.sigma_lower_bound = args["sigma_lower_bound"]
        self.sigma_mult = args["sigma_mult"]
        self.alpha = args["alpha"]
        self.alpha_mult = args["alpha_mult"]
        self.alpha_lower_bound = args["alpha_lower_bound"]
        self.gen_size = args["gen_size"]
        self.rewards = np.zeros(self.gen_size)
        self.theta = np.zeros(self.theta_dim)
        self.best_theta = np.zeros(self.theta_dim)
        self.best_reward = -float("inf") # lowest possible value
        self.forget_best = args["forget_best"]
        self.weight_decay = args["weight_decay"]
        self.rank_fitness = args["rank_fitness"]
        if self.rank_fitness: # use rank rather than fitness numbers
            self.forget_best = True # forget the best one if we use rank fitness

    def ask(self):
        '''returns a collection of model weights'''
        self.epsilon = np.random.randn(self.gen_size, self.theta_dim)
        self.solutions = self.theta.reshape(1, self.theta_dim) + self.epsilon * self.sigma

        return self.solutions

    def tell(self, simulation_results):
        '''updates internal variables based on the results from simulating the results from
           self.ask '''

        # input must be a numpy float array of the correct length
        if len(simulation_results) != self.gen_size:
            print "incorrect length of input"
            return None

        rewards = np.array(simulation_results)

        if self.rank_fitness:
            rewards = compute_centered_ranks(rewards)

        # decay the weights of our neural network using
        if self.weight_decay > 0:
            l2_decay = compute_weight_decay(self.weight_decay, self.solutions)
            rewards += l2_decay

        index_ordering = np.argsort(rewards)[::-1]

        best_reward = rewards[index_ordering[0]]
        best_theta = self.solutions[index_ordering[0]]

        self.curr_best_reward = best_reward
        self.curr_best_theta = best_theta
        # update our best parameters we are storing
        if self.forget_best or (self.curr_best_reward > self.best_reward):
            self.best_theta = best_theta
            self.best_reward = self.curr_best_reward

        # standardize the rewards to have a gaussian distribution
        normalized_rewards = (rewards - np.mean(rewards)) / np.std(rewards)
        change_theta = 1./(self.gen_size*self.sigma)*np.dot(self.epsilon.transpose(), normal

        self.theta += self.alpha * change_theta

        # adapt sigma if not already too small
        if (self.sigma > self.sigma_lower_bound):
            self.sigma *= self.sigma_mult

        # adapt learning rate if not already too small
        if (self.alpha > self.alpha_lower_bound):
            self.alpha *= self.alpha_mult

    def current_param(self):
        return self.curr_best_theta

    def set_theta(self, theta):
        self.theta = np.array(theta)

    def result(self):
        return self.best_theta
```

Training on Grace

To exploit the parallel nature of the problem, I applied for and used the high performance computing cluster, Grace. This allowed for a dramatic speedup of the training procedure of the policy, and eventually led to the solutions that we will list below to the selected environments. I adapted the parallel training procedure used by ESTool, the open source project written by the author of the blogpost I referenced. This code used MPI as a way to communicate between multiple processes each running playouts of the environment simulation. This parallel playout strucutre, which follows the procedure that OpenAI used in their paper, allows us to test many different random mutations of our current best performing paraments while only having to communicate the random seed used to generate the mutation noise. This, of course, dramatically speeds up the amount of time each generation takes to complete. I used the following run script on Grace

```bash
#!/bin/bash
#SBATCH --partition=scavenge
#SBATCH --job-name=bipedal_walker_hardcore_es
#SBATCH --ntasks=64
#SBATCH --ntasks-per-node=1
#SBATCH --mem-per-cpu=6000
#SBATCH --time=12:00:00
#SBATCH --mail-type=ALL
#SBATCH --mail-user=email

source activate amth
pip install mpi4py
python train.py -e 16 -n 64 -t 4 --start_file <model_start_file>
```

What this code script does is tells grace that we want to run on the scavenge partition, which means that we can use any hardware that is not currently being used by someone else; however, we can be prempted if anyone gets scheduled onto the hardware we got assigned to. The reason that this is acceptable is that the training procedure that I adapted from ESTool allows for frequent checkpointing, so we can start where we left off whenever we get preempted. This means we don't have to wait in any fair-share queues unless there is no hardware available. Next, this script tells Grace that we want 64 tasks for our MPI code and we want one process to be running in each of these tasks.

The last three lines of the script simply switch to our relevant conda environment, which includes the open-ai gym, MPI, and other standard python packages that our code depends on. Additionally the last line of the script calls the training procedure which glues together the code that I presented in the report and spins up 64 MPI workers, each averaging their fitness score over 16 episodes (for robustness purposes) computing 4 trials per MPI worker. These were the recommended training settings from ESTool. Lastly, optionally you can include the model start file, which rather than start from scratch every time, allows you to start at your previously found best parameters. I implemented this functionality to allow the code to be run on the scavenge partition.

Results

The results of the experiment were resoundingly positive. I was skeptical of the methods being able to work on the bipedal walking environmen

Conclusion

In my mind before I started the project and before I read OpenAI's paper, I had always seen evolutionary algorithms as cute black-box optimzation methods that did not have serious application in training modern machine learning models;

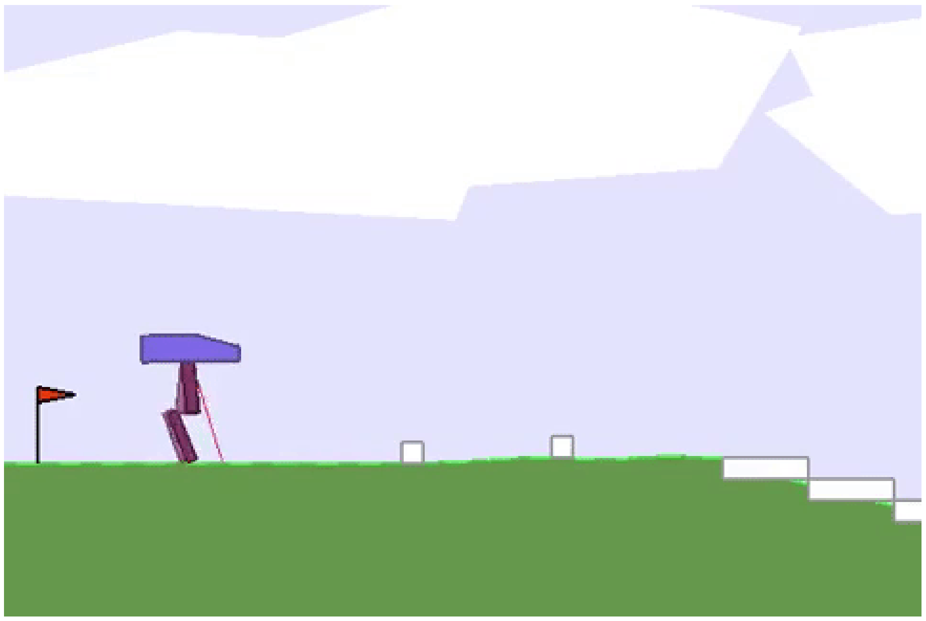

Figure 3: Alt Text

however, as we can see in the results section, these algorithms may actually have a place in the traditional reinforcement learning setting. For parameter vectors that aren't incredibly large, like in the millions, evolutionary algorihtms offer an enticing option to train continuous control agents. The ability to

References

1.
2.
3.